Threat Model of Enterprise Open Source Supply Chains

Date	Participants
Apr 18, 2023	Crob, Victor Lu, Henrik
Apr 25, 2023	Abdullah Garcia, Crob, Victor Lu, Dan, Jonathan, …, Avishay Balter
May 9, 2023	Victor Lu, Itay Katz, Henrik
May 16, 2023	Victor Lu, Itay Katz, Henrik, Amanda Martin, Avishay Balter
May 23, 2023	Victor Lu, Itay Katz, Henrik, Amanda Martin, Avishay Balter
June 13, 2023	Victor Lu, Henrik, Randall, Josh Clements, Georg Kunz, Amanda Martin, Saswata Basu
Jun 20, 2023	Henrik, Anna V
Jun 27, 2023	Henrik, Amanda, Avishay Balter, Anna V, Randall, Victor Lu
July 10, 2023	Henrik
Jul 24, 2023	Henrik, CRob, Victor
Aug 21,2023	Henrik, Victor, Anna
Aug 28,2023	Henrik, Joshua Holmes, Andreas Fehlner (ONNX), Anna (Citi), Jeffrey Borek
Sep 4, 2023	Henrik, Victor
Oct 2, 2023	Yuval Sinay, Anna V, Henrik, CRob, Tracy Ragan, Jon Meadows, Tanner Jones
Oct 16, 2023	Henrik Dana Wang, Ariel Shuper, CRob, Anna

Motivation and Objectives

The goal of this threat model is to identify and describe threats that result from the consumption of open source and other third-party software components in software development processes and infrastructures "typically" observed within large enterprise software development organizations.

https://t.me/learningnets

To this end, the high-level architecture diagram shown below shows typical systems, services and actors involved throughout the software development life cycle of a large development organization. The threat model focuses on consumption of software dependencies as well as the infrastructure used to build solutions for the enterprise.

Once the threat model is built, it can be used to identify gaps in controls and then be mapped to security controls provided by OpenSSF projects and initiatives that can mitigate the identified threats, which in turn gives a practitioner-oriented overview about works conducted at the OpenSSF.

Related Work

- Mitre ATT&CK is a comprehensive knowledge base of generic techniques and tactics used throughout the adversary's lifecycle, from reconnaissance to initial access and persistence, and supports the creation of specific threat models. The technique Supply Chain Compromise (T1195) and its sub-techniques are particularly relevant for this threat model: Sub-technique T1195.002 describes the manipulation of application software, and is considered by the ATT&CK as a technique for getting initial access. In the context of this threat model, however, the manipulation of application software is one of the core concerns. Sub-technique T1195.001 describes the manipulation of software dependencies and tools used throughout an application software's SDLC. In the context of this threat model, this technique relates to all the incoming data flows of 3rd party source and binary code (cf. architecture diagram, highlighted in green). The description is rather high-level and more refined descriptions on how such dependencies can be manipulated can be found in other works such as the S2C2F. Sub-technique T1195.003, compromise of hardware supply chain, is considered out of scope for this threat model.
- SLSA provides a variety of guidelines to increase the security of software supply chains. Those guidelines are structured into four build levels (L0 - L3) with increasing security guarantees. Those build levels can be captured by build attestations, which in turn can be verified by downstream consumers. The architecture diagram used in this threat model contains several elements of the supply chain model underlying SLSA, however, adds some elements that we consider typical for enterprise software development organizations, e.g. a private registry or distribution platform. Moreover, SLSA 1.0 focuses more on threats to the build than on source aspects, which is also reflected by the names of the build levels.
- The seminal papers Backstabber's Knife Collection and Taxonomy of Attacks on Open-Source Software Supply Chains provide an overview of attack vectors at the disposal of attackers to compromise OSS (see here or here for interactive attack tree visualizations). They primarily relate to the incoming data flows of 3rd party source and binary code (cf. architecture diagram, highlighted in green), and less so to threats impacting organizations' internal development environments and processes.
- OpenSSF Secure Supply Chain Consumption Framework (S2C2F) provides guidance on how to securely consume OSS, and targets software development organizations. It

https://t.me/learningnets

comprises 8 practices organized in 4 maturity levels. Again, the threats considered relate more to the incoming data flows of 3rd party source and binary code (cf. architecture diagram, highlighted in green).

- OWASP Software Component Verification Standard (SCVS) provides 6 control groups to define, build, and verify the integrity of software supply chains. Threats are not explicitly mentioned but only implied. Many SCVS controls also counter threats mentioned in this document and are listed in the control section of the respective threat. Other SCVS controls are out of scope for this threat model, e.g. ones related to known-vulnerable OSS or license compliance.OWASP SCVS
- CISA Guidance - <TODO: Explain what's there>
- Mitre Common Attack Pattern Enumeration and Classification (CAPEC) focusses "on application security and describes the common attributes and techniques employed by adversaries to exploit known weaknesses". The supply chain is one out of six CAPEC domains and comprises eight meta attack patterns such as Software Integrity Attack or Modification During Manufacture. Modification During Manufacture deals with attacks on software or hardware components during the manufacturing process, while Software Integrity Attack deals with attacks on the integrity of software after it has been delivered to the end-user.
- The document NIST SP 800-204D, Strategies for Incorporating Software Supply Chain Security in DevSecOps CI/CD Pipelines, presents a detailed breakdown of high-level Software Supply Chain Security practices into CI/CD pipelines. This enables the secure management of the Software Supply Chain. The requirements outlined in the document align closely with the proposed controls. However, the threats that motivate these practices are not extensively explained and are mostly implied.

Methodology

The threat model builds on top of previous works such as SLSA and seminal papers like Backstabber's Knife Collection and a Taxonomy of Attacks on Open-Source Software Supply Chains.

Those works introduce high-level supply chain models comprising SCM, CI/CD and artifact registries, which were extended to cover additional actors and systems considered typical for large enterprise development organizations. Despite those additions, the architecture remains very high-level in order to stay independent of specific tool vendors and to cover as many organizations as possible.

Every threat identified has a name, description, impact, controls and references to existing works and real-world examples. The likelihood of threats is sometimes mentioned in the description, but not provided systematically. The impact is only described qualitatively, but not quantified. If quantification is needed, typical elements contributing to the direct costs of a breach or cyberattack are, for example, remediation costs, legal fees such as litigation or regulatory fines or costs of notifying affected customers and affected stakeholders.

https://t.me/learningnets

Threat Model

Architecture

The following picture illustrates a bird's-eye view of the flow of source code from developers to build systems, the consumption of dependencies during the build, and the production of binary packages that can be consumed by software end-users.

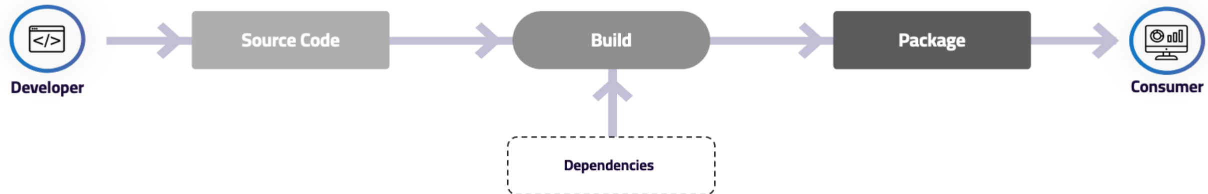

Fig 2: Bird's-eye view on the flow of source code from developers to consumers (taken from here)

The following high-level architecture diagram refines this birds-eye view to show typical systems, services and actors involved throughout the software development life cycle of a larger development organization. The main flow of a given project's source and binary code from left to right is identical. Shown in much greater detail is the consumption of open source dependencies on all involved systems, downloaded from external distribution sites depicted on the right:

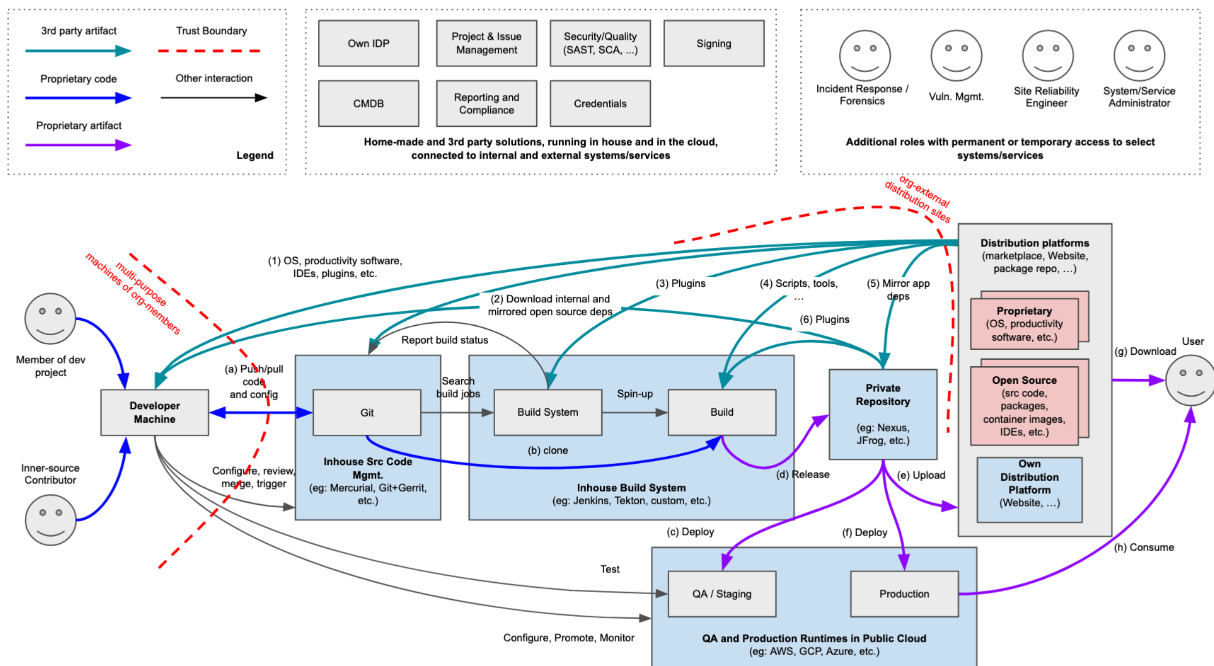

Fig 3: Architecture diagram of a large in-house development organization (taken from here)

The diagram only shows a subset of relevant data flows with a focus on (I) the consumption and use of 3rd party (3P) code by different actors and systems (in **green**), e.g. the download of

https://t.me/learningnets

application dependencies during the build, and (II) the flow of proprietary source code and proprietary artifacts from developer machines to the product's end-user (in **blue** and **violet**).

What is called **developer machine** in the diagram represents an endpoint used not only by developers but also by testers and builders and people with other roles related to software development projects. They are typically multi-purpose systems used for a variety of tasks, not only for developing or testing etc. (SSDF 1.1 PO.5.2 calls them development endpoints). Members of development projects use them to interact with many of the systems and services used throughout the SDLC, e.g. using Web front ends or APIs. For example, they are used to (a) push and pull code from/to the SCM.

The **SCM system**, the **build (CI/CD) system**, the **private repository** (registry) and other development-related systems and services are considered "in house", i.e. hosted and operated by the development organization, and are not internet-facing.

The **end-user** consumes the software product or service either by (g) downloading build artifacts from distribution platforms like download and update sites of commercial vendors, software marketplaces or CDNs, which serve scripts and other Web resources, or by (h) using a cloud service like Software-as-a-Service through Web front ends, APIs or other means.

Additional notes:

- The systems, services and roles depicted in the box "Additional Roles" at the top right of the image interact with many of the systems shown below. However, they are not connected to keep the diagram readable.
- All those systems and services are considered being governed by a corporate security program including, for example, network segregation, endpoint protection capabilities and corporate IDPs.

Trust Boundaries

The following two trust boundaries are considered for this initial version of the threat model:

The first is related to the consumption of 3P components, based on the observation that such components can be compromised in various ways. In particular, we do not only consider application-level dependencies but all kinds of dependencies, e.g. plugins of a build system, shell scripts downloaded and executed during builds, or code snippets taken from public websites and included by the developer in proprietary projects. Those components and code flow from external data sources, e.g. package registries, component marketplaces or developer blogs and forums, to all kinds of internal systems, across their entire stack.

"*Trust, but verify*" should be the general mindset or attitude of developers and development organizations when it comes to consuming open source. The thoroughness of verification depends on the context and the organization's risk appetite, and can range from shallow checks like digest verification to, for example, code-level checks with SAST tools.

https://t.me/learningnets

The second separates development-related endpoints, esp. developer machines, from all other systems running within the organization, which acknowledges that developers, but also testers and builders, are a prime target for supply chain attacks. They cannot only be infected through the above-mentioned data flows, but also through other "classical" vectors, e.g. phishing emails or drive-by downloads, just because those machines are used for a large variety of tasks and therefore have a larger attack surface than specialized systems. Malware running on such endpoints can perform actions on various connected systems on behalf of (authenticated) developers, testers, etc., with the goal to compromise the integrity or confidentiality of software products, e.g. push commits, approve merge requests or exfiltrate Git clones.

However, the current focus on those two trust boundaries does not mean that any interaction or data flow between those can be implicitly trusted, e.g. that authentication and authorization checks can be omitted. In fact, chances are that malicious code or components are eventually downloaded during builds, to internal registries etc.

A future version of this document should therefore make this more explicit and assume trust boundaries around the individual systems, which resonates well with recommendations provided in NIST 800-218, Secure Software Development Framework v1.1: The tasks PO 5.1 and 5.2, for example, suggest to secure developer endpoints and all other development-related systems following a zero trust architecture, which essentially means to introduce strong authentication methods, maintain and enforce fine-grained access control on a need-to-know and least-privilege basis and use secure communication channels for all user-to-system and system-to-system communication.

Assets

Assets of the development organization comprise:

- Proprietary source and binary code: All proprietary artifacts that are part of the final software product as well as intermediate artifacts used and produced throughout product development. This includes, for example, compiled code, resource files or default configuration in regards to the software product, as well as source code, build files, build configurations or pipeline definitions in regards to the development process. When compromised, the development organization may distribute compromised software to their end-users, e.g. one that includes backdoors or alters the application logic.
- System infrastructure configuration: All artifacts that define or configure systems used for product development or product/service operation. This includes, for example, Helm charts, Ansible playbooks, Dockerfiles or other resources managed via CD/GitOps[1][2] workflows. When compromised, the system architecture/configuration of internal or customer-facing systems may become insecure or vulnerable.

---

[1] https://www.redhat.com/en/topics/devops/what-is-gitops

[2] https://thenewstack.io/gitops-on-kubernetes-deciding-between-argo-cd-and-flux/

https://t.me/learningnets

- Mirrored 3rd-party code: All 3rd-party artifacts mirrored or replicated by the development organization that become part of the final software product or are used during product development. This includes, for example, rebundled 3rd-party components or components used for compilation or testing . Once replicated in internal systems, e.g. in internal package repositories or build system caches, 3rd party code must be protected in the same way as proprietary code. Again, a compromise may result in distributing compromised software to end users.
- Credentials and secrets: All the secrets of individuals and systems accessing and modifying above-mentioned assets during product development. This includes, for example, SSH and GPG keys used by developers for authentication and commit signing, or tokens used for programmatic API access.

Scope

In scope:

- Malicious 3rd-party components (both commercial and open source) downloaded and executed during development, build and distribution. The components can contain **“active” malicious code**, which gets executed upon installation, e.g. droppers, crypto miners or ransomware. It can also include intentional vulnerabilities that are technically identical to accidental vulnerabilities introduced by well-intentioned developers. Examples: IDEs, plugins for IDEs or build systems, or application-level dependencies.
- Malicious software and actors that enter the organization’s infrastructure through other means, and which can threaten the development infrastructure through lateral movements. Examples: Phishing, drive-by downloads, or vulnerabilities in internet-facing services operated by the organization.
- Misconfigured or vulnerable systems/services, which can be exploited by malicious actors or software once they run within the premises.

Out of scope:

- Threats resulting from the use or consumption of 3rd-party firmware and hardware, e.g. drivers integrated into the Linux kernel or loaded dynamically (cf. Common Security Threats from the Open Compute Project in regards to 3rd party hardware and firmware)
- Threats related to the development of embedded software, which faces additional complexity due to the manufacturing and transportation of physical goods, which is typically outsourced to separate legal entities that often reside in other legislations.
- Threats that result from the adoption of open-source culture and principles within a commercial development organization (also called inner-source)
- Threats to the integrity or confidentiality of software products after its download by end-users to consumer or customer-owned systems, e.g., on-premise installations operated by customers or end-user machines
- Threats to end-users resulting from the inclusion known-vulnerable OSS in software products
- Threats to the development organization resulting from non-compliance with OSS licenses

https://t.me/learningnets

- Threats requiring physical access to involved systems, e.g. to place hardware dongles or access unencrypted disks

Insider threats are due to malicious actors operating from within the software development organization, e.g. disgruntled employees or contractors. Those actors are granted privileges for systems and services part of the development environment, according to their specific role in products' SDLC, which facilitates the compromise of their integrity and confidentiality. Insider threats have been excluded for the first version of this threat model, but are considered part of the backlog for future iterations. However, the consideration that developer machines can be compromised by malware, which in turn can act on behalf of (authenticated) developers, already covers insider threats to some extent.

Threat or risk prioritization largely depends on product and organizational context, e.g., the legal and regulatory environment of the organization and its products, the kind of data processed by the software product, etc. which is why it is not covered in this document.

Threats

Consumption of 3rd Party (3P) code or components

The threats listed in this section relate to the incoming data flows of 3P code and components from external sources to various systems of the development organizations (highlighted in green in the above diagram).

They result in the consumption of malicious code or components on those systems, which can give attackers initial access to the affected systems, or become part of the final software product shipped to customers or deployed in production.

They resemble and partly extend the threats mentioned in, for example, the Common OSS Supply Chain Threats mentioned by S2C2F or the attack vectors provided in the paper Taxonomy of Attacks on Open-Source Software Supply Chains.

Name (ID)	Compromise a legitimate upstream project (C-1)
Description	An attacker may compromise a legitimate upstream project to infect software products or systems that are dependent on the project artifacts, directly or indirectly, and consume them automatically or manually. This threat relates to all incoming data flows of 3rd party components, no matter whether they are consumed on developer machines, build systems, etc. This threat also has many sub-threats related to how the upstream project is compromised, e.g., through malicious pull requests, the compromise of its build environment, maintainers going rogue, etc.

https://t.me/learningnets

Potential Impact	Initial access to systems of the development organization and/or systems of product end-users. This can happen to “active” malicious code such as reverse shells or droppers, or through the addition of deliberate vulnerabilities ending up in the organization's software product, which can be exploited by attackers once a product is operational in the end-users’ environment. Once an attacker got initial access to systems of the development organization, he can move laterally using techniques like “Taint Shared Content” (ATT&CK T1080), e.g. through the compromise of shared artifact caches hosted on a build server, or “Remote Services” (T1021), e.g. by performing actions as a logged-on developer.
Controls	- Pin versions of app-level dependency declarations to avoid automated update (and integrate dependency management tools into your SCM to keep receiving security fixes) - Verify digests or signatures to detect attacks on the distribution mechanism of upstream components, e.g. package registries or CDNs - Contain the impact of malicious code run during builds by using dedicated, isolated and ephemeral build systems (cf. SLSA Build Platform and OWASP SCVS V3) - Establish a vetting process that covers individual component versions (cf. OWASP SCVS V5) - Build components from the sources to mitigate attacks on build or distribution mechanisms of upstream components
Example(s)	- Compromised npm package: event-stream (2018) - Codecov incident (2021) - Open source developer corrupts widely-used libraries, affecting tons of projects (2022)
References	- AV-001 in the seminal paper Taxonomy of Attacks on Open-Source Software Supply Chains - Confirmed malicious packages reported by OSV

Name (ID)	Create name confusion (C-2)

https://t.me/learningnets

Description	An attacker may craft new component names that resemble names of legitimate open-source or system components, suggest trustworthy authors or play with common naming patterns in different languages or ecosystems. Malicious components with such names are then deployed on source code or package repositories waiting to be downloaded by victim users or developers. Since the package name does not yet exist in the respective repository, the deployment can be done very easily, without interfering with any legitimate packages, including the one(s) that inspired the new name. Name confusion attacks such as typosquatting or brand-jacking can affect every package registry and have significantly increased in number after 2019/2020. Common targets include app-level package registries like PyPI and npm, but marketplaces for Visual Studio Code plugins and PowerShell modules have also been affected more recently.
Potential Impact	Initial access to systems of the development organization, typically the developer machine or the build system. Typical malware examples contain dropper logic or exfiltrate sensitive information such as environment variables. However, more elaborate attacks could also seek to become part of the software product in order to infect end-users (which requires the attacker to copy or mimic the functionality of the legitimitate project to make the component pass functional tests).
Controls	- Establish a vetting process for all kinds of 3rd party components installed on dev-related systems, covering both direct and indirect dependencies - Enforce vetted allow lists through internal binary registries that mirror approved components and deny access to non-vetted components (cf. OWASP SCVS V4)
Example	- PyPI Python repository hit by typosquatting sneak attack (2017) - PowerShell Gallery susceptible to typosquatting and other package-management attacks (2023) - Malicious VSCode plugins (2023)
References	- AV-200 in the seminal paper Taxonomy of Attacks on Open-Source Software Supply Chains - Confirmed malicious packages reported by OSV

Name (ID)	Missing, incomplete or inconsistent vetting process (C-3)

https://t.me/learningnets

Description	An organization may lack a vetting or review process for 3rd party components altogether, or it may be incomplete (in regards to the types of components covered, the checks performed or the information collected). In case different departments are responsible for different parts of the hardware/software/cloud/AI stack, vetting or review processes may be inconsistent.
Potential Impact	All of this may result in the download and execution of non-reviewed and non-approved components. This can allow attackers to get initial access to the development organization and/or systems of product end-users through the installation/execution/use of malicious components.
Controls	- Make sure that the vetting process covers the entire stack of all the systems involved in the development process, direct and indirect dependencies, as well as individual component versions. Allow lists should not only mention component names and versions, but also comprise digests, signatures, download sites and - for open source components - the respective SCM (cf. SLSA Verifying artifacts, OWASP SCVS V5).
Example	- Malicious VSCode plugins

Name (ID)	Infect during review (C-4)
Description	An attacker may prepare and include payloads in 3rd party components that trigger when being examined with trusted developer tools.
Potential Impact	Initial access to the system used for verifying and vetting 3rd party components.
Controls	- Perform component reviews in sandbox environments
References	- Trusted Developer Utilities Proxy Execution (ATT&CK T1117) - ldd arbitrary code execution

Name (ID)	Deploy on non-standard distribution channels (C-5)
Description	An attacker may deploy infected 3rd party components on non-standard distribution sites, e.g. rogue download mirrors, in the hope that they get downloaded and installed from there.

https://t.me/learningnets

Potential Impact	Initial access to systems of the development organization.
Controls	- Include authorized download sites, digests and signatures in components' vetting information - Verify digests or signatures before consumption - Prevent access of build systems to non-approved network addresses - Enforce vetted allow lists through internal binary registries that mirror approved components and deny access to non-vetted components (cf. OWASP SCVS V4)
Example	- XCodeGhost (2015)

Name (ID)	Misconfigured package managers (C-6)
Description	Mis-configured package managers do not download from trusted sources (e.g., internal package registries) but directly from the Internet.
Potential Impact	Developers install malicious components
Controls	- Control: Alway verify before installation (and the verification depends on the risk appetite of the resp. Organization, distinguish integrity checks and "deeper" checks) - "Trust but verify" is a general mindset or attitude that developers should have, maybe we include this when discussing about controls
Example	- The Maven package manager requires the modification of settings.xml to consider internal registries. - CVE-2021-26291

Name (ID)	Promote Vulnerable Code in Dev Communities (C-7)
Description	An attacker places and promotes/upvotes malicious or deliberately vulnerable code, e.g. in blog posts, via developer communities such as StackOverflow or through sample code on GitHub, hoping that the code is picked up by developers and copied verbatim into their development projects.
Potential Impact	Vulnerable code is included in a software product. Depending on the platform or Web site used, attackers can get context on how and

https://t.me/learningnets

	where the code is integrated, e.g. due to user interaction, associated user accounts, social-engineering, etc.
Controls	- Raise developer awareness that code shared on StackOverflow (or generated by LLMs) can be vulnerable (both accidentally or deliberately) - Incentivize or enforce code reviews, e.g. through branch protection in SCM workflows - Perform security tests for 1st party code
References	- OpenSSF Education SIG: Goal: Identify sources of insecure code snippets - A Study of C/C++ Code Weaknesses on Stack Overflow (2021)

Infrastructure Threats

This section lists threats that are mostly due to malicious actors having gained initial access through any of the threats listed in the previous section. Or - more generally - what can go wrong with data flows and interactions happening within the development organization.

Developer machine

Name (ID)	Push malicious commit with legit developer account (I-1)
Description	A malicious actor, e.g. an external attacker having obtained initial access to a developer machine or an insider, may push a malicious commit (with malicious code or an intentional vulnerability) to the internal SCM as the authenticated and legitimate developer. The commit may carry a valid signature, using the key material of the developer machine, or impersonate other developers (e.g. by setting Git user.email and user.name) in order to go unnoticed or receive an approval (in case of protected branches).
System Component	Developer machine
Potential Impact	The malicious commit enters the SCM containing proprietary code. The commit may be picked up by build pipelines, become part of build results, and be distributed or deployed at later stages (all those steps can happen automatically or through manual interaction). Depending on how actively the proprietary component is developed, such changes can go unnoticed for a longer period of time, e.g. in case of software under maintenance (which could still be managed through automated build pipelines).

https://t.me/learningnets

Controls	- Establish SCM access control and keep authorizations up-to-date in order to prevent malicious contributions - Disable force push - Enforce review workflows for sensitive branches - Enforce commit signatures - Verify consistency of GitHub commiter/pusher information to detect user.email spoofing
Examples	- Changes to Git commit workflow (PHP, 2021) - Pushing code to GitHub as Linus Torvalds (2013) - When Dependabot Contributes Malicious Code (2023)
References	- AV-301 in the seminal paper Taxonomy of Attacks on Open-Source Software Supply Chains - AV-700 in the seminal paper Taxonomy of Attacks on Open-Source Software Supply Chains

Name (ID)	Approve malicious PR with legit developer account (I-2)
Description	A malicious actor, e.g. an external attacker having obtained initial access to a developer machine or an insider, may approve a malicious pull/merge request (created by another malicious actor) as the authenticated developer.
System Component	Developer machine
Potential Impact	A malicious commit pushed by another actor, e.g. infected developer machine, is merged (manually or automatically) and enters the build pipeline. The selection of an unsuspicious committer and approver identities can be based on the git history or information obtained from a collaboration tool like Gerrit
Controls	- Establish SCM access control and keep authorizations up-to-date in order to prevent malicious contributions

Name (ID)	Exfiltrate system or pipeline secrets (I-3)
Description	An attacker having obtained initial access to a system of the development organization, e.g. developer machine or build system, may exfiltrate secrets, e.g. ssh and GPG keys stored on disk or API tokens provided through environment variables. This technique is frequently used by malicious packages spread through name

https://t.me/learningnets

	confusion (see threat "Create name confusion"), which often upload such secrets to Discord or Telegram chats or custom Web services
Potential Impact	Lateral movement to other systems using compromised credentials
Controls	- Don't allow untrusted pipeline jobs/steps to use trusted credentials (e.g. jobs created from forked repositories) and reduce availability/scope of credentials within pipelines - Don't use user-provided context (e.g. PR titles) in scripts - Use dedicated credential stores (instead of the Jenkins credentials, for example) - Limit network connectivity of build systems to prevent exfiltration
Example	- Divide and Hide: How malicious code lived on PyPI for 3 months (PyPI, 2023)
References	- ATT&CK T1567, Exfiltration Over Web Service - Using a Jenkinsfile - Automatic Security Assessment of GitHub Actions Workflows (2022)

Name (ID)	Exfiltrate sensitive repository content (I-4)
Description	Malicious actors exfiltrate confidential information that got accidentally or intentionally stored in the versioning control system. Example information: - Production and deployment information - Proprietary source code - Test data from production systems - Secrets accidentally committed into repo
System Component	Developer machine or build system
Potential Impact	- Reconnaissance - Lateral movements - IP theft
Controls	- Run secret detection solutions, e.g. TruffleHog, Credential Digger or Gitleaks, to detect past leaks and in pre-commit hooks to prevent future ones - Limit network connectivity of build systems to prevent exfiltration - ??

https://t.me/learningnets

References	

Name (ID)	Blackmail developer (I-5)
Description	Malware running on the developer machine blackmails the developer, e.g., using confidential data found on the machine, or threatening to encrypt or delete data
System Component	Developer machine
Potential Impact	Developers are blackmailed to perform malicious activities in the interest of the attacker, e.g. share credentials or code, or inject malicious code into source code or binary artifacts.
Controls	- Incentivize or enforce code reviews, e.g. through branch protection in SCM workflows - Perform security tests for 1st party code - Don't build artifacts on developer machines - ??
References	- AV-601 in the seminal paper Taxonomy of Attacks on Open-Source Software Supply Chains

Name (ID)	Redirect Traffic to Attacker-controller Resources (I-6)
Description	Redirect developer traffic to attacker-controller resources, which could be inside or outside of the company's network, e.g. through DNS spoofing or similar.
System Component	All
Potential Impact	Tampered resources become part of the project and/or become executed during the build.
Controls	- Include authorized download sites, digests and signatures in components' vetting information - Verify digests or signatures before consumption
References	- Fortify: Attacking the Build through Cross-Build Injection (2007)

Name (ID)	Tamper with shared resources (I-7)

https://t.me/learningnets

Description	A malicious actor tampers with shared resources, e.g. system components, local or remote caches. Examples: - Local artifact caches like the Maven M2 repository on developer machines or build systems. - Shared system components like compilers or runtimes.
Potential impact	Compromised resources used in different contexts or projects spread malicious code across development projects.
Controls	- Use project-specific and ephemeral environments - Restrict access of project resources to shared system resources - Verify integrity of resources used by different build steps
Examples	
References	- SLSA Build Platform Isolation Strength - NIST SP 800-204D

Name (ID)	Tamper with project resources (I-8)
Description	A malicious actor tampers with project resources like source code, compiled code, manifest files with dependency declarations etc. Example: - A malicious test or test dependency tampers with source or compiled code of the project [1, p.32]
Potential impact	- Compromised project resources are packaged and distributed to end-users.
Controls	- Verify the integrity of individual inputs to different build steps - Isolate individual build steps and/or restrict their access to project resources (test execution, for example, should not have write-access to project resources used as input in later build steps, e.g. source code)
Examples	- SolarWinds (2020)
References	1. Jeff Williams: Enterprise Java Rootkits (2009) 2. NIST SP 800-204D

https://t.me/learningnets

- Tampering with backups of dev machines → malicious components get active once the backup is restored (out of scope, as the tampering with backups probably relates to insider threats)
-

https://t.me/learningnets

Source code management system

Scope:

- In scope are open source version control systems, esp. Git, even if use-cases involving very large files/binaries are not covered by Git or Git LFS, e.g., video game development. Mercurial has been added as another example to the diagram.
- Furthermore, we focus on a central Git setup, which is supposedly the typical setup at bigger development orgs, where one Git is per convention the leading instance.
- Out of scope are commercial solutions like GitHub Enterprise, GitLab etc.

Name (ID)	Git server misconfiguration
Description	Insecure configuration of Git Server itself could potentially allow an attacker with access internally to escalate privileges and access / manipulate source code or install malicious software
System Component	System and VCS server + components (e.g. Git and Gerrit)
Likelihood	
Potential Impact	- Unauthorized installation of server or VCS components, e.g. Git hooks or Gerrit plugins - Change of authorization settings or approval workflows, e.g. the removal of approval requirements for change requests, or the change/addition of authorized approvers

Name (ID)	Rewrite Git history
Description	Rewrite Git history on the “central” Git instance (not through commits, but through tampering with the Git log stored on the disk of the Git server). Depending on how actively the component is developed, such changes can go unnoticed for a longer period of time, e.g. in case of software under maintenance, which could still be managed automatically through build pipelines.
System Component	Git server
Likelihood	Low (requires deep understanding of Git internals)
Potential Impact	Rewritten Git history is picked up by the build pipeline, which builds/distributes/deploys it automatically.
Controls	- Project/organization: Evaluate running Git decentrally, which could mitigate this threat - Consumer: Pin versions to prevent automated updates to malicious versions produced using this vector.

https://t.me/learningnets

Name (ID)	
Description	Leak of information about personnel, roles and responsibilities or development workflows, branches, products and product lines. Such information could be stored in Gerrit, workflow or collaboration systems on top of Git or project management systems. A lot of information can be obtained simply by crawling the Git history, e.g. key developers of subsystems or committers' email addresses.
System Component	Git server, Developer machine, or any other machine connected to the Git server
Likelihood	High
Potential Impact	Information can be used for further OSINT reconnaissance and used to craft plausible or convincing attacks (e.g. social engineering attacks or malicious commits).
Controls	

Name (ID)	Git server vulnerability
Description	Exploit vulnerabilities in the Git server itself, or any of its plugins/extensions
System Component	
Likelihood	
Potential Impact	
Controls	
Examples	PHP's Git server hacked to add backdoors to PHP source code

Threat

- Components that require admin access on the Git server have the same problems as components installed by devs on their machines (and therefore are subject to the same vettings processes etc. and corresponding threats).

https://t.me/learningnets

Existing guidance

- CNCF work group (link from Victor)
- Check SLSA and S2C2F recommendations on VCS setup and configuration
- OSSF SCM Best Practices
- CNCF Best Practices Guide

Differences regarding threats on the developer machine

- Attacks targeting admins or developers differ in the sense that malware running on developer machines have access to the source code. Attacks on admins can try to use admin permissions for the services and systems managed by the admins, e.g. by exploiting known vulnerabilities.

Build System

Directly from Att&ck:

- https://attack.mitre.org/techniques/T1587/002/ - Develop Capabilities: Code Signing Certificates - make like certs - get higher trust
- T1525 Implant Internal Image - build system could be made to run malicious image
- Invalid Code Signature
- Code Signing Certificates - steal certificated used in code signing
- Code Signing Policy Modification - changing the policy on how code signing is used - perhaps on dev machine?

Name (ID)	Cache poisoning
Description	Tampering with local caches (or other shared resources) such as the local M2 repository, e.g. through physical access (insider threat) or other malware running on that machine with the developer's privileges (e.g., a test dependency checked less rigorously tampering with other system components)
System Component	Build system
Likelihood	?
Potential Impact	Infected components are consumed in the builds of artifacts consuming the same cache
Controls	Ephemeral build environments …
Examples / References	- WikiSym 2013: Security of public continuous integration services - SolarWinds (if we consider more generally the case that resources have been tampered with)

https://t.me/learningnets

Build

Private repository

Name (ID)	Upload / override artifacts in internal registry
Description	Adversaries use the registry's API or UI to upload or override artifacts stored in the private registry, e.g. mirrored 3rd-party components or internal components.
System component	Private Registry
Potential Impact	Once available in the internal registry, infected components are downloaded in build processes, where they become included in the final product, or alter the build inputs or results.
Controls	- Setup and maintain the registry's access control mechanism - Consumers should verify digests of downloaded artifacts during the build of their projects - Write access logs - Internal artifacts should be signed and stored together with the signature in the registry - Consumers should verify signatures of internal artifacts during the build of their projects
Example	An open source component that got mirrored in the internal registry is overridden/replaced by a malicious actor.

Staging / Production Environments

Name (ID)	Download artifacts from the wrong registry during the deployment step. Changing the registry location.
Description	Deployment pulls artifacts from the incorrect registry causing incorrect

https://t.me/learningnets

	versions to be installed.
Example	- In Maven, for example, one can change registry locations in the various settings.xml files. - For Docker, there's a JSON file where the list of registry mirrors is defined. - For other tools, there may be environment variables.

Appendix

Dataflows

The following tables explain the colored data flows of the architecture diagram in more detail.

Flows of 3rd-Party Code

Flows of 3rd-Party Code (into and within the organization)		
1	**Source**	Org-external software distribution platforms Examples: Download servers, package repositories, Git repositories or GitHub release artifacts
	Sink	Org-internal development environments (or "development endpoints", cf. SSDF 1.1 PO.5) Examples: Individual developer machines or shared environments such as internal cloud machines and VMs
	Purpose	Support developers throughout the local development process, e.g. - IDEs and IDE plugins - SDKs and runtimes - Application dependencies - Etc. Comprises binaries and source code (built locally).
2	**Source**	Org-internal software repositories hosting 3rd party software that gets manually or automatically mirrored, and which potentially went through some vetting process

https://t.me/learningnets

	Sink	Org-internal development environments (see 1 for details)
	Purpose	Support developers (See 1 for details)
3	**Source**	Org-external software distribution platforms
	Sink	Build systems
	Purpose	Creation of a functional build system tailored for the organization. Examples: - The build system itself, e.g., a Jenkins controller - Jenkins plugins downloaded from dedicated marketplaces - Runtimes (depending on where jobs run)
	Comments	For regulated orgs, flows 3 and 4 originate from some sort of private repositories, where all software is vetted and stored.
4	**Source**	Org-external software distribution platforms
	Sink	The environment running the build job Examples: - Jenkins nodes/agents - Kubernetes pods or Docker containers (e.g. configured as Jenkins agents)
	Purpose	Make tools and scripts available to build jobs that produce the proprietary 1st-party artifacts. Download can happen during the job execution or when creating the node or container image. Examples: - curl-bash of shell scripts (cf. codecov attack). - Infrastructure as code, maybe infrastructure used for load testing?, which maybe requires adding another dataflow to QA environments, search for additional input from end-user group?
	Comments	For regulated orgs, flows 3 and 4 originate from some sort of private repositories, where all software is vetted and stored.
5	**Source**	Org-external software distribution platforms
	Sink	Org-internal software repositories
	Purpose	Replicate 3rd-party code internally to make/keep it available for internal consumption. Replication can happen manually or automatically, and include or not a vetting or review process. Examples: - App-level dependencies
6	**Source**	Org-internal software repositories
	Sink	The environment running the build job
	Purpose	Examples: - 1st and (mirrored) 3rd party container images spin-up during the build job - 1st and (mirrored) 3rd party app-level dependencies such as Maven artifacts or Python wheels

https://t.me/learningnets

<table>
<tr><td rowspan="3">7</td><td>Source</td><td>Org-external software distribution platforms</td></tr>
<tr><td>Sink</td><td>Source code management</td></tr>
<tr><td>Purpose</td><td>Example components:
- Git server, LFS, omnibor, access control (PHP attack from 2020)
- Post-commit hooks (shell scripts deployed in server directories)</td></tr>
</table>

Flows of 1st-party code

<table>
<tr><th colspan="3">Flows of 1st-party code
(within the organization and to consumers)</th></tr>
<tr><td rowspan="3">b</td><td>Source</td><td>Versioning Control System</td></tr>
<tr><td>Sink</td><td>The CI/CD environment running the build job

Examples:
- Jenkins nodes/agents
- Kubernetes pods or Docker containers (e.g. configured as Jenkins agents)</td></tr>
<tr><td>Purpose</td><td>Project source code is cloned to the build environment for testing and build purposes</td></tr>
<tr><td rowspan="3">f</td><td>Source</td><td>Internal test, staging or QA environment</td></tr>
<tr><td>Sink</td><td>Production environment</td></tr>
<tr><td>Purpose</td><td>Upon successful tests, the cloud service is deployed into a production environment</td></tr>
</table>

https://t.me/learningnets